TP d'initiation à Python et à de la programmation

Laurent Guéguen & Claire Guillet

22 janvier 2016

1 Types et opérateurs de base

Python est un langage de programmation qui paraît interprété (c'est-à-dire que l'on peut exécuter des commandes directement par une interface) mais qui est en fait pré-compilé pour être exécuté via une machine virtuelle (comme JAVA). Ceci le rend à la fois très pratique d'utilisation et très rapide.

Ce langage est disponible sur le site :

`http://www.python.org/`

Pour accéder à l'interface de Python, il suffit de taper la commande `python` dans un éditeur de commandes. Le prompt est désormais remplacé par >>>.

Pour quitter l'interface, taper `Contrôle D`. On peut aussi remonter et descendre dans les commandes grace aux flèches ↑ et ↓.

1.1 Variables

Dans une ligne, tout ce qui est à droite du dièse '`#`' est du commentaire non pris en compte. Ainsi dans les exemples suivants, on ne demande pas de taper ces commentaires.

Exécuter les lignes suivantes :

```
>>> 2+3
>>> 2+3-5
>>> 2*(1-(3+5))
>>> 2/3        # division entière
>>> 2.0/3      # division de flottants
>>>
>>> a=3        # attribution
>>> a
>>> print a
>>> a+=5
>>> a
>>> type(a)
>>> a/3        # division entière
>>> float(a)
>>> float(a)/3 # division réelle
```

Python est un langage *typé*, c'est à dire qu'une variable appartient à un type donné (`int, string, list, ...`). Dans un type, les variables ne peuvent

prendre qu'une certaine gamme de valeurs. Ainsi, `int` concerne les valeurs entières entre -2^{31} et $2^{31}-1$. Ce typage se fait à la volée, quand une valeur est attribuée à une variable.

Si on continue l'exemple :

```
>>> b=float(a)  # conversion de type
>>> print b
>>> type(b)
>>>
>>> c=str(b)    # conversion de type
>>> print c
>>> type(c)
>>>
>>> del c       # destruction de c
>>> c           # ERREUR: c n'existe plus
```

Pour les types de base, l'opérateur `print` permet d'afficher la valeur des variables, mais peut être omis.

N'importe quel mot qui commence par une lettre ou _ et qui n'est pas un mot réservé du langage peut être un nom de variable.

1.2 Chaînes de caractères (type `str`)

Une chaîne de caractères (string) correspond à un tableau de caractères, lesquels sont accessibles par leur indice. Il s'agit d'un moyen très économique en mémoire pour stocker des données, mais qui ne peut être modifié simplement.

```
>>> s='acgt'
>>> s          # la valeur
>>> print s    # meilleure présentation
>>> len(s)     # longueur de la cha\^ine
>>> s[0]
>>> s[3]
>>> s[4]       # ERREUR: débordement de tableau
>>> s[-1]
>>> s[-4]
>>> s[1:3]
>>> s[2:]
>>> s[:3]
>>> s[:4:2]
>>> s[-2::-1]
>>> s[0]='b'       # ERREUR: on ne peut changer le contenu
>>> print s+'att'  # concaténation
>>> s+='b'         # crée une NOUVELLE cha\^ine, référencée par s
>>> print s
>>> print 2*s
>>> a='rac2 vaut ' +str(1.41421)
>>> print a
```

Une chaîne de caractères peut aussi être délimitée par " et """. Le dernier délimiteur autorise les retours chariot dans les strings. Ainsi,

```
>>> s="ac'g"+"""tg
... na"""
>>> s        # la string n'est pas interprétée
>>> print s  # la string est interprétée
```

La notation `a.f()` est la façon standard d'appeler, pour l'*instance* `a` de la *classe* `str` la *méthode* `f`. Ces termes sont importants, et deviendront courants en programmation orientée-objet. Par exemple :

```
>>> s='acbDEFghI'
>>> s.swapcase()   # inverse majuscules et minuscules
>>> s.__len__()
>>> len(s)
```

Par exemple, `len(a)` est juste une commodité de notation pour la commande `a.__len__()`, et de même `a[2]` est une commodité pour `a.__getitem__(2)`. Le vérifier.

1.3 Aide

Pour accéder à toutes les fonctions disponibles pour une string, il suffit de taper :

```
>>> help(str)
```

Pour quitter l'aide, on tape `q`. On navigue dans l'aide grâce aux flèches, ou les lettres `f, b, p` ainsi que l'espace. Lorsque la page d'aide est trop grande, il peut être utile de sauvegarder celle-ci dans un fichier en tapant `s`.

Dans la description des fonctions, la présence de crochets signifie que les arguments qui les suivent sont en option. Lorsqu'on veut fournir ces arguments, il ne faut pas mettre les crochets.

Vérifier dans cette aide la présence des méthodes `__len__` et `__getitem__()`.

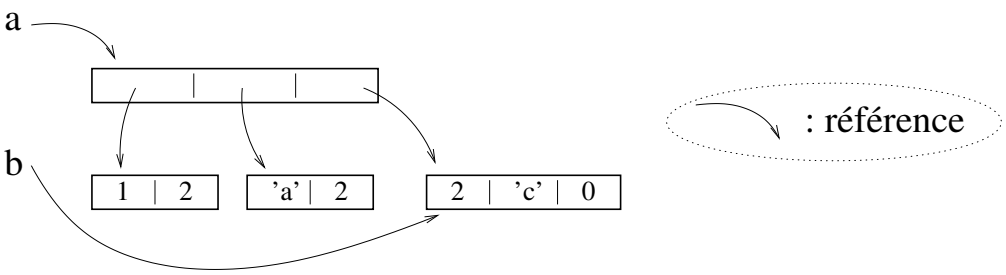
Exercices :

1. `s='acgtaagta'` ;
2. Quintupler `s` ;
3. Mettre toutes les lettres de `s` en majuscules ;
4. Remplacer dans `s` les 3 premières occurences de `'AAC'` par `'TTA'` ;
5. Compter dans `s` les occurences de `'AG'`.

1.4 Listes (type `list`)

Les listes disposent aussi des opérateurs `len, [], +, +=`. Essayer

```
>>> range(10)    # commande qui crée une liste
>>> range(8,1,-1)
>>> range(0,19,4)
>>>
>>> x=[1.2,4,'a',6]  # liste
>>> type(x)
>>> x[:3]
>>> x[1:4:2]
>>> x.append(2)      # ajout à la fin de la liste
>>> print x
>>> a=[[1,2],['a',4],[2,'c',0]]  # liste de listes
>>> print a[1][1]
>>> a[1]='all'
>>> print a
>>> b=a[2]           # gestion des références
>>> b[0]='tt'        #
>>> print a          # comprendre ce qui se passe
>>> b=[9]            #
>>> print a          #
```



`id` donne la valeur de la référence de l'objet. Ainsi :

```
>>> a=[1,2]
>>> id(a)
>>> id(a[0])
>>> b=a[0]
>>> id(b)
>>> b=3
>>> id(b)
>>> b=a
>>> id(b)            # Même objet
>>> b=a[:]           # VRAI opérateur de copie
>>> id(b)
>>> a=[1,[2,3]]
>>> b=a[:]
```

```
>>> b[1][0]=4
>>> a
```

La notion de référence est fondamentale pour comprendre le fonctionnement de Python, et pour programmer.

Exercices (à chaque ligne, une seule commande suffit) :

1. Créer la chaîne de caractères `s='acggtgctta'` ;
2. Insérer des espaces (' ') entre les lettres de `s` ;
3. Enlever les espaces qui sont aux extrémités de `s` ;
4. créer une liste `l` constituée de la liste des lettres de `s` ;
5. quintupler `l` ;
6. construire la liste `m`, qui sera `l` renversée ;
7. trier l par ordre alphabétique croissant.

1.5 Dictionnaires (type `dict`)

Il s'agit d'objets dans lesquels des valeurs (de n'importe quel type) sont assignées à des clefs (de type `int, float, long, complex`, ou même `str` ou `tup`).

```
>>> a={2:4,'b':[1,2]}
>>> a['b']
>>> a[3+4j]=5              # 3+4j est un nombre complexe
>>> a
>>> a['c']                 # ERREUR : clef inconnue
>>> len(a)
>>> a.keys()
>>> a.values()
>>> b={}                   # construit un dictionnaire vide
>>> id(a)
>>> b=a
>>> id(b)
>>> b=a.copy()             # copie profonde
>>> id(b)
```

En dépit de sa lourdeur apparente, le dictionnaire est un type extrêmement efficace en python, et il ne faut pas craindre d'en user et abuser, même avec des clefs artificielles (du genre `1, 2, 3,`...).

Exercices :

1. Construire le dictionnaire `d` correspondant au tableau suivant :

clefs	valeurs
ac	23
tt	32
ga	10
aa	5

2. ajouter à `d` la valeur 14 pour la clef `'cg'` ;
3. enlever de `d` la clef `'tt'` et mettre la valeur associée dans la variable `x` ;
4. lister les clefs de `d`.

1.6 Modules

Des bibliothèques de fonctions, appelées modules, sont disponibles en Python. Le chargement d'un tel module se fait ainsi :

```
>>> import math     # chargement du module
>>> math.log(2)
>>> help(math)
>>> from random import randint   # chargement d'un objet
>>> randint(1,30)
>>> randint(1,30)
>>> randint(1,30)
>>> random.randint(1,30)    # erreur
>>> from time import *      # chargement de tous les objets
>>> asctime()
>>>
>>> import commands               # module qui permet d'exécuter
>>>                               #  des commandes extérieures
>>> s=commands.getoutput("ls -l") # exécute 'ls -l' et récupère
>>> print s                       #   la sortie
```

1.7 Fichiers (type `file`)

La gestion des fichiers est très simple, aussi bien en lecture qu'en écriture. Dans tout langage, il faut comprendre les fichiers comme des flux de données, en écriture comme en lecture. Ainsi, par exemple, l'écriture des données se fait successivement à partir du début du fichier.

```
>>> f=open('toto','w')   # ouverture du fichier toto en ecriture
>>> f.write('Bonjour le monde\n')    # ecriture d'une phrase.
>>>                                  # \n signifie retour-chariot
>>> f.write('Comment vas-tu?\n')
>>> f.close()                    # ne pas oublier
```

Éditer par ailleurs le fichier `toto`, puis dans python :

```
>>> f=open('toto','r')   # ouverture du fichier toto en lecture
>>> l=f.readline()       # lecture d'une ligne
>>> l
>>> print l              # print interprete \n dans la ligne
>>> print f.readline()   # lecture de la ligne SUIVANTE
>>> print f.readline()   # lecture de la ligne SUIVANTE
```

```
>>> print f.readline()   # lecture de la ligne SUIVANTE vide
>>> print f.readline()   # lecture de la ligne SUIVANTE vide
>>> f.close()            # ne pas oublier
>>>
>>> f=open('toto','a')   # ajout dans toto
>>> f.write('Tres bien, je programme.')
>>> f.close()
```

Pour lire un fichier, il existe aussi la fonction `readlines()` (avec un `s`) qui renvoie une liste où chaque ligne du fichier forme un élément. C'est donc une façon rapide de lire toutes les lignes d'un fichier d'un coup. Toutefois, comme cette méthode amène à charger tout le fichier d'un coup, elle est aussi très coûteuse en mémoire et si le fichier est gros cela peut avoir un impact sur les performances du code. Il est donc conseillé de préférer `readline()` à `readlines()`, surtout lorsqu'on souhaite lire de gros fichiers.

2 Structures de contrôle

La conception d'un programme passe par l'utilisation de conditions, boucles, fonctions, que l'on appelle *structures de contrôle*.

La syntaxe Python veut que le domaine d'une clause (tel `if`, `for`, etc) est déterminé par une indentation, réalisée par une tabulation. Un début de structure de contrôle est reconnu par le fait que sa ligne termine par un deux-points ' :'. Des structures imbriquées entrainent des indentations cumulées, à savoir plusieurs tabulations. On sort d'une structure en commençant la ligne d'après par une tabulation de moins.

Dans une structure de contrôle, le prompt devient : '...'. Dans l'interface, pour sortir de toutes les structures, il suffit de taper `entrée` sur une ligne vide.

Ainsi, dans les exemples suivants, **il est important de bien respecter les lignes blanches et les tabulation**.

2.1 Conditions

Il s'agit d'effectuer des ordres sous une condition. Pour cela, il faut effectuer un *test*, et si le résultat de ce test est vrai, les ordres voulus sont effectués. Éventuellement, dans le cas contraire, d'autres ordres sont effectués.

Pour effectuer des tests, on dispose de nombreux opérateurs qui retournent `True` ou `False`.

```
>>> a=2
>>> a==2   # test d'égalité
>>> a!=2   # test de différence
>>> 2>3
>>> 2>=2
>>> 3<=4
>>> not a>=4                # négation
>>> a in [3,2,4]            # test d'appartenance
>>> (a in [0,2]) and (2<5)  # et logique
>>> (3>5) or (2<5)          # ou logique
```

Le résultat d'un test est évalué pour conditionner une exécution par le mot réservé `if`. Dans l'exemple suivant, `b` prend la valeur 6 si `a` est plus petit que la valeur initiale de `b`.

```
>>> a=5
```

```
>>> b=3
>>> if a<b:
...     print 'b est plus grand que a'
...     b=6
...
```

Le mot réservé `else` décide d'un comportement si le test est faux. Par exemple, pour l'affichage du maximum entre deux valeurs :

```
>>> a=5
>>> b=3
>>> if a<b:
...     c=0
...     print b
... else:
...     c=1
...     print a
...
>>> print c
```

Avec le mot réservé `elif` (pour `else if`), on peut cumuler des conditions exclusives :

```
>>> x='a'
>>> if x=='c':
...     print 'cyt'
... elif x=='g':
...     print 'gua'
... elif x=='t':
...     print 'thy'
... else:
...     print 'ade'
...
```

2.2 Boucles

Il s'agit d'exécuter plusieurs fois la même succession de commandes.

Par le mot réservé `for`, on peut procéder à des itérations finies :

```
>>> s='accgcacgaagt'
>>> for i in s:
...     print i

>>> l=len(s); print l     # par le ; on peut cumuler les commandes
>>> for i in range(l):
...     if s[i]=='a':
...         print 'ade en position ' + str(i)
...
```

La lecture de fichiers peut-être faite par un `for` :
On peut aussi faire :

```
>>> f=open('toto','r')   # ouverture du fichier toto en lecture
>>> for l in f:          # lecture d'une ligne
...     print l
...
>>> f.close()
```

Par le mot réservé `while`, pour « tant que », on procède à des itérations conditionnelles :

```
>>> s='gctagctaag'
>>> i=0
>>> while i<len(s):
...   if s[i]=='a':
...       break     # fait sortir de la boucle
...   i+=1
...
>>> print 'première adénine en position' + str(i)
```

Exercices :

1. Créer la string `s="hg18.7"` ;
2. Avec une boucle, créer la liste des caractères de `s`, dans le sens inverse ;
3. Avec une boucle, identifier la première position de `s` qui est un chiffre (commande `str.isdigit()`.

2.3 Fonctions

La définition de fonctions se fait par le mot réservé `def`.

```
>>> def fonc(a,b):        # déf de la fonction fonc à 2 arguments
...      print type(a), type(b)
...      return a+b       # return renvoie une valeur
...                       #         et SORT de la fonction
>>>
>>> fonc(2,3)          # appel de la fonction
>>> fonc('ab','cd')
>>> fonc('ab',2)       # arguments pas cohérents
>>>
>>> def f(a):
...     print a
...     return a
...     print a+1      # ne sert à rien, n'est pas lue
...
>>> b=f(3)
```

Les variables qui sont définies dans une fonction ne sont connues que dans leur contexte. À l'extérieur de la définition de la fonction, elles ne sont pas définies. On parle d'*espace de noms* (spacename) de la fonction. La *portée* (scope) de la fonction est l'ensemble des variables qui y sont disponibles, qu'elles aient été définies à l'intérieur ou à l'extérieur de celle-ci. Ainsi :

```
>>> def f(aa):       # aa est un parametre local
...      bb=4        # bb est une variable locale
...      return aa*bb
...
>>> f(3)
>>> aa               # erreur: aa pas définie
>>> bb               # erreur: bb pas définie
>>> a=5
>>> def g(x):
...     return x*a   # a est une variable globale
...                  #  donc dans la portée de g
>>> g(5)
```

Il existe différents espaces de noms, tels que les classes, méthodes et modules. Pour ces espaces de noms, on parle aussi de portée.

Exercices :

1. Soit la fonction :

   ```
   >>> def occ(a,s):
   ...     l=len(s)
   ...     v=[]
   ...     for i in range(l):
   ...         if s[i]==a:
   ...            v.append(i)
   ...     return v
   ...
   >>> x=occ('g','attggatag')
   >>> print x
   ```

   Simuler à la main l'exécution de `occ` sur l'exemple donné ;
2. Écrire une fonction qui modifie une string `s` en une liste de strings, qui correspondent aux mots de `s` (i.e. séparés par des blancs) ;
3. Écrire une fonction qui séparé une string fait d'une alternance de caractères et de chiffres (commme `'ac12gct34'`) une liste alternant les mots et les valeurs (retourne `['ac',12,'gct',34]`) ;

2.4 Fichier source

En pratique, lors de la conception d'un programme, on définit les fonctions dans un fichier source, et on exécute via l'interface.

Ainsi, récupérer et éditer le fichier source python

`http://pbil.univ-lyon1.fr/members/gueguen/lib/TP_Python/lecture.py`

Dans l'éditeur de commandes, taper `python lecture.py` permet d'exécuter les commandes définies dans le fichier `lecture.py`. Mais cela ne permet pas de rentrer dans l'interface python.

Dans l'interface, on peut aussi exécuter un fichier sous python, en tapant

```
>>> import lecture            # sans le .py
>>> lecture.lit('toto')       # Exécution sur le fichier 'toto'
```

Dans le fichier `lecture.py`, la fonction `lit` lit et affiche successivement les lignes d'un fichier, dont le nom est donné en argument. Lorsque la dernière ligne de ce fichier a été lue, `readline` retourne la chaîne vide. Par abus, cette chaîne mise en condition retourne `False`. On aurait aussi pu écrire

`while l!='':`

Lorsqu'un fichier a été modifié, on le recharge en tapant `reload`. Ainsi, dans notre exemple, on tape : `reload(lecture)`.

3 Analyse de séquences nucléotidiques

Le travail de ce tutoriel consistera à analyser des séquences alignées de trois espèces : Homme (*Homo sapiens*), Chimpanzé (*Pan troglodytes*), Macaque rhésus (*Macaca mulatta*).

À l'aide d'un navigateur, récupérer le fichier *en tant que texte* à l'adresse :
`http://pbil.univ-lyon1.fr/members/gueguen/lib/TP_Python/ENm010.maf`
Éditer ce fichier pour voir comment il est conçu.

Un alignement est constitué de séquences, chacune représentée par un dictionnaire, dont les clefs qui nous intéressent sont [1] :

esp l'espèce : type `str` (hg, panTro, MacRhe) ;

vers la version : type `int` (par exemple 18 pour hg18.7) ;

chrom le chromosome : type `str` (par exemple '7' pour hg18.7) ou '0' s'il n'est pas connu) ;

pos la position sur la chromosome : type `int` (ou 0 si elle n'est pas connue) ;

nbl le nombre de lettres alignées : type `int` (i.e. sans les gaps '-') ;

sens le sens : type `str` ('+' ou '-') ;

lgchr la longueur du chromosome : type `int` (ou 0 si elle n'est pas connue) ;

seq la séquence nucléotidique : type `str` (avec les gaps '-').

Ainsi, un alignement sera un dictionnaire dont les clefs sont les noms d'espèces, et les clefs des dictionnaires correspondant aux séquences avec les clefs décrites ci-dessus.

3.1 Mise en bouche

3.1.1 Lecture

Dans le fichier `lit_maf.py`, la fonction `lit_maf` permet de lire un tel fichier. Le fichier est là :
`http://pbil.univ-lyon1.fr/members/gueguen/lib/TP_Python/lit_maf.py`
Exécuter cette fonction. Comprendre ce qu'elle retourne (et en option, ou plus tard, comment elle fonctionne).

1. On peut trouver une explication complète de ce format sur
`http://genome.ucsc.edu/FAQ/FAQformat#format5`

```
import lit_maf
m=lit_maf.lit_maf("ENm010.maf")
```

3.1.2 Quelques fonctions pour la manipulation

Voici des fonctions à programmer qui s'avéreront utiles par la suite. Ensuite, libre à vous d'en concevoir d'autres en fonction de vos besoins.

lettre retourne, depuis un alignement et un entier n, le dictionnaire qui à chaque nom d'espèce associe la lettre à la position n dans l'alignement, ou bien `'-'` si la position n n'est pas valide ;

sequence retourne, depuis un alignement et deux entiers n et m , le dictionnaire dont les clefs sont les noms d'espèces et les valeurs les séquences correspondant au segment de la position n à la position m (exclue) dans l'alignement. Les positions non valides du segment (par exemple m trop grand) sont omises (comme pour l'opérateur `[:]` de `str`) ;

Recherche de mots

On peut chercher des sites de fixation potentiels de facteurs de transcription dans des séquences (souvent en amont du promoteur). En reprenant les séquences des alignements précédents, nous allons chercher les sites de signal d'initiation de la transcription `GEN_INI`. Ces sites de fixation ne sont pas dans les séquences répétées.

- Écrire une fonction `find` qui dans une séquence d'un alignement retourne la liste des positions (en tenant compte des indels) où se trouve un mot donné. Ne pas oublier que les indels ne sont pas des lettres standard, et donc doivent être omis dans cette recherche ;
- Sachant que le facteur de transcription peut reconnaître ces deux séquences (`GATTGGCA` et `AAGTGAGT`), recherchez les occurences dans les séquences ;
- Pour chacune de ces occurences, affichez l'alignement local correspondant. Ces séquences sont-elles conservées ?

Pour attester que ces sites sont vraiment des sites promoteurs, il faudrait ensuite croiser ces informations avec celles obtenues pour d'autres facteurs de transcriptions.

3.2 Comparaisons de séquences

Dans la suite, vous serez amenés à construire des méthodes par vous même, pour répondre aux problèmes posés. N'hésitez pas !

Cette partie peut être abordée après le chapitre 4, qui permet de les programmer plus facilement sous forme d'objets, moyennant un investissement méthodologique *a priori*.

3.2.1 Éléments répétés

Dans ces séquences, les parties qui sont en minuscules correspondent aux éléments répétés (éléments transposables, motifs répétés, etc).

— Définir une méthode `lg_sr` qui retourne la longueur totale des éléments répétés dans une séquence ;
— Sur la donnée du fichier `maf`, comparer entre les trois génomes la proportion d'éléments répétés ;

  Normalement, un élément répété dans une séquence correspond soit à une insertion, soit à un élément répété dans une autre séquence alignée. Si cette règle n'est pas respectée, cela signifie qu'un élément répété n'a pas été détecté dans une séquence. On se sert alors du fait qu'il a été détecté comme élément répété dans une séquence alignée pour corriger cette cette erreur.
— Écrire une fonction qui corrige les séquences de manière à ce que cette règle soit respectée ;
— En tenant compte de cette correction, comparer de nouveau les proportions d'éléments répétés dans les génomes ;
— Écrire une fonction `masque` qui renvoie un dictionnaire ou un `Alignement` dans lequel tous les éléments répétés sont masqués (i.e. dont les lettres sont remplacées par des `N`).

3.2.2 Distances

Une évaluation grossière de la distance évolutive entre deux séquences consiste à mesurer la proportion de sites différents entre ces séquences.

— Concevoir une fonction qui retourne un dictionnaire dont les clefs sont les couples d'espèces, et les valeurs les nombres de divergences entre ces séquences ;
— En utilisant cette fonction, concevoir une fonction qui retourne le même type d'objet, mais dont les valeurs sont le nombre de divergences entre les génomes.
— Les résultats de cette fonction sont-ils cohérents avec la phylogénie des espèces ?

3.2.3 Reconstruction par parcimonie

Un intérêt de ces alignements est de reconstruire la séquence ancestrale de l'homme et du chimpanzé (que l'on appellera HC), en utilisant le macaque comme groupe externe.

Il s'agit d'écrire une fonction `parciseq_Hs_Pt` qui retourne une séquence reconstruite par parcimonie. Le principe de ce mode de reconstruction est de trouver le scenario évolutif qui minimise le nombre de substitutions. Si ce scenario est unique, il permet d'inférer une lettre au nœud ancestral.

— Réfléchir, en fonction des triplets possibles, aux différents cas qui permettent d'inférer par parcimonie la base ancestrale homme-chimpanzé ;

— Écrire une fonction `parcisite` qui pour un triplet de lettres (voir la fonction 3.1.2), retourne la lettre ancestrale homme-chimpanzé si elle peut être identifiée, '-' sinon.
— Écrire la fonction `parciseq_Hs_Pt` [2] ;
— Comparer, entre les nucléotides, les processus de substitution sur les deux branches, qui relient HC et chacune des deux espèces filles ;
— On s'intéresse à l'évolution des dinucléotides `CN` (avec `N` qui vaut n'importe quelle lettre) dans la séquence HC. Comparer leur évolution dans les deux branches, en fonction de la lettre qui est après le `C` [3].
— Faire la même étude au niveau de la divergence Homme-Chimpanzé-Macaque.

2. Quand on construit une `string` de cette manière, il est plus économique de construire la liste des lettres, puis de faire un `join`.

3. Ne pas oublier que les `CN` sont aussi présents dans les `NG`.

4 Programmation orientée-objet

Pour une bonne présentation de la programmation orientée-objet, voir le site : `http://www.commentcamarche.net/poo/poointro.php3`

Il s'agit de concevoir des classes, équivalents plus élaborés des types, munies de caractéristiques (les attributs) et de fonctionnalités (les méthodes). Ces classes faciliteront considérablement la manipulation des données.

4.1 Classe Seq_ali

Dans le fichier `seq_ali.py`, créer la classe `Seq_ali` se fait par :

```
class Seq_ali:
```

Lorsque l'on crée un objet, il faut dire comment le construire, c'est-à-dire quels arguments il a, et quelles sont leurs valeurs initiales. Ceci est fait par le constructeur

```
class Seq_ali:
    def __init__(self):     # constructeur
       self.esp=""     # espece (hg, panTro, MacRhe)
       self.vers=0     # version
       self.chrom="0"  # le chromosome
       self.pos=0      # la position sur le chromosome
       self.nbl=0  # le nombre de lettres alignées
       self.sens="+"   # le sens
       self.lgchr=0 # la longueur du chromosome
       self.seq=""     # la séquence nucléotidique (avec les gaps)

    def ecrit(self):
       s="s\t"+self.esp+str(self.vers)+"."
       s+=self.chrom+"\t"+str(self.pos)+"\t"+str(self.nbl)+"\t"
       s+=self.sens+"\t"+str(self.lgchr)+"\t"+self.seq
       print s
```

Respecter l'indentation, car on est dans le domaine ouvert par `class Seq_ali:`

Attention : toute définition de méthode a pour premier argument `self` et les attributs sont appelés par `self.` en préfixe.

Ensuite, dans l'interface python, taper :

```
>>> import seq_ali        # le module a pour nom celui du fichier, moins .py
>>> ex=seq_ali.Seq_ali()  # on crée un objet Seq_ali, via l'appel de
>>>                       #  Seq_ali.__init__(self)
>>> ex.__class__.__name__  # nom de la classe de ex
>>> isinstance(ex,seq_ali.Seq_ali)
>>>
>>> ex.esp="hg"      # on donne la valeur "hg" à l'attribut esp de ex
>>> ex.vers=18       # ------------------  18  ----------- vers de ex
>>> ex.ecrit()       # on appelle la méthode écrit de ex
>>>                  #   (noter la disparition du self)
```

Dans la définition de la classe, `self` correspond à l'objet construit lui-même (self), vu de l'intérieur. Vu de l'extérieur, l'**instance** a un nom, et on appelle ses attributs et méthodes grâce à ce nom. Ainsi, si on continue dans le fichier `seq_ali.py` :

```
    def copie(self, s):
       "Copie la Seq_ali s dans celle-ci"
       self.esp=s.esp      # copie de l'espece de s
       self.vers=s.vers
       self.chrom=s.chrom
       self.pos=s.pos
       self.nbl=s.nbl
       self.sens=s.sens
       self.lgchr=s.lgchr
       self.seq=s.seq
```

et donc :

```
>>> reload(seq_ali)
>>> ex=seq_ali.Seq_ali()     # REconstruire l'objet selon la
>>> ex.vers=13               #   nouvelle version
>>> ex2=seq_ali.Seq_ali()
>>> ex2.copie(ex)
>>> ex2.ecrit()
```

La ligne de texte sous la déclaration de la fonction est une information affichée par `help`.

```
>>> reload(seq_ali)     # permet de relire seq_ali.py
>>> help(seq_ali)
```

La fonction usuelle d'afficher le contenu d'une variable est `print` :

```
>>> print ex2
```

affiche l'identité de `ex2`, mais pas son contenu, car `print` fait appel à `__str__` si elle la voit pour afficher le contenu.

— Remplacer dans `Seq_ali` la méthode `ecrit` par `__str__` qui retourne une string au format `maf` de l'objet :

```
def __str__(self):
  s="s\t"+self.esp+str(self.vers)+"."
  s+=self.chrom+"\t"+str(self.pos)+"\t"+str(self.nbl)+"\t"
  s+=self.sens+"\t"+str(self.lgchr)+"\t"+self.seq
  return s
```

On vérifie que cette méthode fonctionne bien dans l'interface en tapant, si l'objet `Seq_ali` s'appelle `x` :

```
>>> str(ex)      # retourne la string
>>> print ex     # écrit la string
```

— Écrire la méthode `__getitem__(self,n)` qui retourne pour la position `n` la lettre correspondante de la séquence, ou `'-'` si la position n'est pas valide.
La méthode `__getitem__` est, comme `__str__`, une méthode particulière par sa syntaxe : il s'agit de l'opérateur `[]` (voir par exemple l'aide de `str`). Ainsi, avec cette fonction, sur une `Seq_ali s`, il est possible d'appeler `s[2]`.

4.2 Encapsulation

Il s'agit d'un principe fondamental de développement orienté-objet, qui permet qu'un programme puisse être développé avec fiabilité par différents agents, et qu'il puisse être ultérieurement amélioré par un agent sans compromettre l'ensemble.

Dans le cas de python, on définit deux types d'attributs et de méthodes, privés et publics [1]. Les données privées sont inaccessibles par l'extérieur de la classe, alors que les données publiques sont visibles par tout le monde.

L'encapsulation consiste à rendre privées les attributs des classes, et à les rendre accessibles de l'extérieur uniquement par l'intermédiaire de méthodes *ad hoc*. Cela permet éventuellement

— de protéger l'accès à ces attributs contre des mauvaises utilisations ;
— de simplifier l'utilisation de la classe en ce que l'utilisateur ne doit pas chercher comment les attributs sont gérés, et en construisant des méthodes avec des noms et utilisations *pertinentes* ;
— de modifier la classe (en particulier le type de donnée des attributs) de façon transparente pour l'extérieur, du moment que les méthodes d'accès garantissent le même résultat.

En python, une donnée (attribut ou méthode) est privée si elle commence par `__` (double souligné). Dans ce cas, elle n'est accessible qu'à l'intérieur de la classe.

1. Dans d'autres langages, on parle en outre de membres protégés.

— Ainsi, par exemple, dans `Seq_ali`, remplacer `esp` par `__esp`, et essayer d'accéder par l'interface à cette donnée.
— Pour accéder à cet attribut, écrire une méthode `esp(self)` qui retourne la valeur de cet attribut, ainsi qu'une méthode `set_esp(self,n)` qui donne à cet attribut la valeur `n`.
Il faudrait faire de même avec les autres attributs de `Seq_ali`.
— Recopier et adapter la fonction `lit_seq(ligne)` du fichier `lit_maf.py` en une méthode `lit_seq(self, ligne)` qui remplit de façon adéquate les attributs de `Seq_ali` depuis une string au format `maf`. Tester cette fonction avec une ligne type :

```
s.lit_seq("s hg19.chr7    115810521 37 + 159138663 CTTTAC")
print s
```

4.3 Autres classes

4.3.1 Classe `Alignement`

On veut concevoir, dans le fichier `alignement.py`, une classe `Alignement` qui gère un alignement de `Seq_ali`.

Cette classe sera munie de deux attributs :

dico un dictionnaire qui associe nom d'espèce et `Seq_ali` ;

score un score (float).

Définir une telle classe, dont les attributs seront privés. Comme cette classe doit connaître la classe `Seq_ali`, écrire en début de fichier :

```
import seq_ali
```

Munir cette classe des méthodes :

`__str__` qui retourne une chaîne de caractères au format `maf` correspondant à l'objet ;

`score` **et** `set_score` qui retournent et définissent la valeur de score ;

`copie` qui prend un `Alignement` en argument, et remplit les attributs de `self` avec les valeurs des attributs de celui-ci ;

`__getitem__` qui prend un nom d'espèce en argument, et retourne le `Seq_ali` correspondant ;

`lettre` qui prend une position en argument et retourne le dictionnaire qui à chaque espèce associe la lettre de la séquence correspondante en cette position (relative à l'alignement), ou la lettre `'-'` si la position n'est pas valide ;

Recopier et adapter la fonction `lit_ali(fichier)` du fichier `lit_maf.py` en une méthode `lit_ali(self, fichier)` qui lit à partir d'un fichier ouvert en lecture et positionné au début d'un alignement, les `Seq_ali` correspondants, et les stocke de façon adéquate.

4.3.2 Classe `Lalignement`

Enfin, on veut gérer l'ensemble des informations disponibles dans un fichier d'alignements : cela sera fait dans la classe `Lalignement` du fichier `lalignement.py`.

Créer cette classe, avec comme attributs :

liste une liste d'objets `Alignement` ;

intro une chaîne de caractères ;

La chaîne de caractères `intro` correspond aux premières lignes du fichier `maf` utilisé pour construire le `Lalignement`.

Dans cette classe, écrire les méthodes :

— Recopier et adapter la fonction `lit_maf(nom_fichier)` du fichier `lit_maf.py` en une méthode `lit_maf(self, nom_fichier)` qui lit un fichier au format `maf` et remplit les attributs avec les valeurs correspondantes ;

`__str__` qui retourne la chaîne de caractères correspondant aux lignes d'`intro`, suivies des strings correspondant aux différents alignements, de manière à afficher les données au format `maf`.

4.4 Héritage

Le principe de l'héritage est de définir une hiérarchie entre des classes, les plus spécifiques héritant des plus générales. Voir le site
`http://www.commentcamarche.net/poo/heritage.php3`

La classe `Seq_ali` définit les séquences au sein d'alignements, donc pourvues d'indels, et d'attributs spécifiques tels que `nbe_ali`. C'est pour cela qu'elles sont au format `maf`. En revanche, une séquence seule est plutôt enregistrée au format `fasta`, et ne nécessite pas tous les attributs et méthodes de `Seq_ali`.

C'est pour cela que l'on va revoir la définition de `Seq_ali`, pour la faire hériter d'une classe `Sequence`. Cette classe est dans le fichier
`http://pbil.univ-lyon1.fr/members/gueguen/lib/TP_Python/sequence.py`

Écrire les méthodes de `Sequence` :

`__str__` qui retourne une `string` de la séquence au format fasta ;

`__getslice__` qui retourne une `string` correspondant à une sous-séquence de la séquence.

Ainsi, en faisant hériter `Seq_ali` de `Sequence`, il n'est plus besoin de définir dans `Seq_ali` des attributs et méthodes qui sont déjà définis dans `Sequence`. Le fichier `seq_ali.py` commence par :

```
from sequence import Sequence

class Seq_ali(Sequence):  # héritage

  def __init__(self):
      Sequence.__init__(self)   # Appel du constructeur de Sequence
      self.__esp=''
```

```
        self.__vers=0
        .
        .
        .
```

Ensuite, comme les méthodes `seq`, `set_seq`,... de `Seq_ali` on été copiées à l'identique dans `Sequence`, on peut enlever leur définition de `seq_ali.py`.

Les attributs et méthodes publics de la classe mère sont transmis directement à la classe fille, et donc peuvent être utilisés par cette dernière comme s'ils y étaient définis. En revanche, les membres privés (par ex `__seq`) ne peuvent être appelés directement, mais sous le nom masqué (`_Sequence__seq`).

— Modifier la fonction `__str__` de `Seq_ali` pour tenir compte du déplacement des attributs dans `Sequence`.

Ainsi, on peut construire :

```
>>> import seq_ali
>>> s=seq_ali.Seq_ali()
>>> s.set_seq('ACGGCTCGAT') # méthode définie dans Sequence
>>> len(s)
>>> print s  # méthode définie dans Seq_Ali: surcharge
>>> print Sequence.__str__(self)  # appel de la méthode __str__
>>>                               #   de Sequence
>>> isInstance(s,seq_ali.Seq_ali)
>>> isInstance(s,seq_ali.Sequence)
```

— Déplacer dans `Sequence` les méthodes de `Seq_ali` qui y ont leur place.

4.5 Travail sur les segments

4.5.1 Classes de segments

Parmi les séquences que l'on vient d'étudier, il y a des morceaux de séquences spécifiques : les îlots CpG (ou CGI), et les CDS. S'ils ont en commun d'être des segments sur une séquence, ils ont aussi des spécificités. Ainsi, nous allons définir une classe `Segment`, de laquelle des classes `CGI` et `Exon` vont dériver. Ensuite, un `CDS` sera une liste d'`Exon`.

Les coordonnées de ces segments, sur le chromosome 7 humain, sont là :
`http://pbil.univ-lyon1.fr/members/gueguen/lib/TP_Python/CDS_encode.txt`

et

`http://pbil.univ-lyon1.fr/members/gueguen/lib/TP_Python/CGI_encode.txt`

Récupérer aussi les fichiers `segment.py`, `cds.py`, `cgi.py` et `exon.py` au même endroit.

Attention : les positions dans les segments correspondent à la séquence **sans gap**, donc il faut bien faire attention aux décalages lorsqu'on gère ces segments au niveau des alignements.

— Comprendre la définition de ces classes ;
— Écrire une fonction qui lit un fichier au format de `CGI_encode.txt`, et qui retourne une liste de `CGI`. Faire de même avec le fichier `CDS_encode.txt` ;
— Écrire dans `Segment` une méthode `seq_ali` qui, sur un paramètre `Lalignement`, retourne l'`Alignement` dans lequel est ce `Segment`, `None` si un tel `Alignement` n'existe pas ;
— Écrire une classe `Align_seg` qui hérite d'`Alignement`, et qui a comme attribut supplémentaire une liste de `Segment` sur le génome de l'homme couverts par cet `Alignement` ;
— Écrire dans `Align_seq` une méthode qui assigne comme `Segment` les séquences répétées identifiées chez l'homme dans cet alignement.

4.5.2 Îlots CpG

Il s'agit d'étudier si les processus évolutifs sont les mêmes dans les îlots CpG par rapport au reste de la séquence.

— Écrire une méthode de `Lalignement` qui, depuis une liste de `CGI`, retourne un `Lalignement` dont la liste des `Alignement` est constituée des `Align_seg` qui ont un îlot CpG ;
— Comme au 3.2.3, reconstruire pour chaque `Seq_align` par parcimonie la séquence ancestrale Homme-Chimpanzé, puis étudier les patrons de substitution – sur les nucléotides et sur les dinucléotides CN –, le long de la branche qui mène à l'homme, en faisant la différence entre les îlots et les non-îlots ;
— Est-ce que l'on voit des différences au niveau des séquences répétées ? On peut faire la différence entre les séquences répétées identifiées à la fois chez l'homme et le chimpanzé, ou uniquement chez l'homme.
— Les îlots ont été référencés sur le génome humain. Est-ce que patrons de substitution sont les mêmes dans tous les îlots, le long de la branche qui mène au chimpanzé ?
— En option, étudier ces patrons le long de la branche qui mène au macaque.

4.5.3 CDS

Faire la même chose que précédemment, mais cette fois pour les CDS. L'objectif est de comparer les taux de substitution entre nucléotides dans les différentes phases des CDS.

4.5.4 îlots « propres »

Les contraintes cumulées sur les îlots et les exons complique la lisibilité des phénomènes. Ainsi, refaire le travail du 4.5.2, en prenant bien soin d'enlever des îlots CpG les parties codantes et les séquences répétées.

4.5.5 Élements répétés

Étudier l'influence de l'insertion d'un élément répété dans un îlot CpG : est-ce qu'il acquiet le patron de sustitution de l'îlot dans lequel il est ? Est-ce que

le patron de substitution au bord d'un îlot est déplacé en raison de la présence d'un élément répété ? Est-ce que l'on peut utiliser la répétition de ces éléments pour aider cette analyse ?

Expressions régulières

Les expressions régulières (ER) sont un moyen de rechercher des motifs, plus ou moins déterminés, dans des chaînes de caractères. Elles sont très utilisées sous UNIX, par exemple avec les fonctions de type `grep`, `awk`, `sed`, ou dans les langages de scripts. Leur manipulation est notamment un des atouts majeurs du langage `Perl`.

Un module d'expressions régulières, le module `re`, a également été développé sous Python. N'en abusez-pas, il y a parfois des solutions plus simples ou en tout cas plus efficaces pour résoudre votre problème. Cependant elles peuvent être très utiles dans certains cas.

Typiquement les ER peuvent être utilisées pour répondre aux questions du type : "est-ce que ma chaîne de caractères correspond à mon motif" ou "est-ce que mon motif est reconnu a un endroit dans la chaîne" ? On peut éventuellement les utiliser pour modifier une chaîne de caractères ou la couper de différentes manières.

4.6 Comment utiliser les ER en Python ?

Une ER est un objet spécifique qu'il faut compiler avant utilisation. Les ER sont utilisées en Python avec le module `re`. Avant toutes choses, commencez donc par taper :

```
>>> import re
```

4.6.1 Compiler les ER

Pour pouvoir utiliser un motif en Python, il faut tout d'abord le compiler en un objet spécifique à Python. Cela se fait au moyen de la fonction `re.compile()`.

```
>>> p = re.compile('ab')  # l'ER est le mot    ab
>>> print p
<_sre.SRE_Pattern object at 0x7e5c0>
```

La fonction `compile` peut s'utiliser avec des options (*flags*). Voici celles qui peuvent vous être utiles :

— `I` ou `IGNORECASE` : les classes de caractères et chaînes littérales vont correspondre à la chaîne de manière insensible à la casse.

— `M` ou `MULTILINE` : De manière habituelle, les méta-caractères `^` et `\$` ne vont faire correspondre les caractères qu'au début ou à la fin de la chaîne. Avec cette option, `^` et `\$` vont aussi faire correspondre les caractères qui suivent (qui précèdent) un retour à la ligne.
— `S` ou `DOTALL` : le méta-caractère `.` correspond aussi aux retours à la ligne (sinon, il correspond à tout sauf le retour à la ligne).

Pour les utiliser :

```
>>> p = re.compile('ab', re.I)
```

Nota bene : Certains meta-caractères, comme le `\b`, peuvent avoir une autre signification en Python. Pour que le motif soit compilé correctement, il convient de faire précéder la chaîne de caractère de la lettre `r` (ce qui crée une "chaîne brute" (*raw string*) en Python : les caractères ne sont pas interprétés).

Par exemple :

```
>>> p = re.compile(r'\bbo*a\b')
```

N'hésitez pas à utiliser une "chaîne brute" même si ce n'est pas indispensable.

4.6.2 Rechercher les correspondances

Qu'est-ce qu'on peut faire une fois que l'on a un motif compilé ? Il existe quelques fonctions qui nous permettent de répondre aux questions que l'on se posait dans l'introduction de cette partie.

Mon ER se trouve-t-elle dans la chaîne que je regarde ? Il existe plusieurs fonctions pour cela (`help(re)` pour en avoir la liste). Entre autres :

— `match()` détermine si une ER se trouve au début d'une chaîne ;
— `search()` parcourt toute la chaîne en cherchant une position qui correspond à l'ER. Si plusieurs telles positions existent, cette fonction renvoie la plus petite.

Si rien n'est trouvé, ces fonctions renvoient `None`. Sinon, elles renvoient un autre objet spécifique à Python, qui possède les attributs suivants :

— `group()` renvoie la chaîne qui correspond à l'ER ;
— `start()` renvoie la position de départ de la correspondance ;
— `end()` la position de fin ;
— `span()` un tuple contenant les positions de départ et de fin.

Sur un petit exemple :

```
>>> p = re.compile('ab')
>>> m = p.match('abracadabra')
>>> m
<_sre.SRE_Match object at 0x26c5d0>
>>> m.group()
'ab'
>>> m = p.match('attention')
>>> m
```

```
None
>>> m = p.search('abracadabra')
>>> m.group()
'ab'
>>> m.span()
(0, 2)
```

`finditer` retourne un itérateur qui permet de récupérer toutes les occurences **non recouvrantes** d'une ER :

```
>>> m = p.finditer('abracadabra')
>>> for x in m:
...     print x.span()
...
>>>
```

4.7 Les ER de base

On utilise les ER en définissant le motif que l'on recherche. Ce motif contient deux types de caractères : ceux qui correspondent à eux-mêmes (les caractères alpha-numériques de `a` à `z` et les chiffres par exemple) et les caractères spéciaux (ou méta-caractères).

4.7.1 Premiers caractères spéciaux

- Le point `.` correspond à n'importe quel caractère sauf les fins de ligne [2].
- Les crochets ouvrant et fermant `[` et `]` signifient *ou* pour des caractères. Ainsi `[ag]` correspond à la lettre `a` ou à la lettre `g` ;
  À l'intérieur de ces crochets, les autres méta-caractères perdent leur caractère "spécial" et se comportent comme les autres caractères.
  Par simplification, les crochets permettent aussi de définir une classe de caractères successifs. Par exemple, si je veux les chiffres `0`, `1` et `2` : `[012]` ou `[0-2]` ; ou encore toutes les lettres minuscules : `[a-z]`.
- Pour définir une classe par *complémentation*, par exemple tous les caractères sauf `5`, on utilise l'accent circonflexe associé aux crochets : `[^5]`.
- Certains caractères spéciaux représentent un ensemble de caractères, et peuvent être utilisés tels quels même dans les classes de caractère. Par exemple :
  - `\s` correspond à tous les caractères d'espacement (`\t`, `\n`, `r`, `f`, et `v`)
  - `\d` correspond à tous les chiffres (équivalent de la classe `[1-9]`)
  - `\w` correpond à tous les caractères alpha-numériques (c'est l'équivalent de la classe `[a-zA-Z1-9]`)
- La contre-oblique `\` permet de traiter comme caractères normaux les caractères spéciaux. Par exemple, si on veut inclure dans notre motif le crochet, il faudra écrire `\[`, ou `\\` pour `\`.

2. dans un mode alternatif (`'re.DOTALL'`) il peut correspondre à tout même les fins de ligne - cf. plus haut

4.7.2 Si on veut répéter notre motif

— `*` indique que l'ER précédente doit être vue zéro fois ou plus, au lieu d'une seule fois (par exemple `bo*a` va correspondre à `ba` (pas de `o`), `boa` (1 `o`), `boooa` (3 `o`), etc...).
— `+` indique que l'ER précédente doit être répétée une fois ou plus ;
— `?` indique que l'ER prédédente doit être répétée zéro ou une fois ;
— Pour définir des répétitions plus complexes, on peut utiliser les accolades. Ainsi, `x{m,n}` indique que le caractère `x` doit être répété au moins `m` fois et au plus `n`.
— Une ER peut être incluse entre deux parenthèses ; ainsi `(bo)+a` correspond à `boa`, `boboa`, `boboboa`, ...

4.7.3 De nouveaux méta-caractères

Certains méta-caractères ne correspondent pas à un ou plusieurs caractères, mais affirment quelque chose quant aux caractères environnants. En voici quelques exemples :

— Le caractère `|` correspond au 'ou' logique entre mots. Par exemple, si `A` et `B` sont deux ER, `A|B` va correspondre avec les chaînes qui possèdent `A` ou `B`.
— Le caractère `^`, utilisé hors des classes de caractères, ne permet de reconnaitre le caractère suivant qu'en début de ligne (le début de la chaîne et les caractères après les retours à la ligne).
— Le caractère `\$` ne permet de reconnaître le(s) caractère(s) qu'en fin de ligne (le caractère avant le retour à la ligne ou la fin de la chaîne).
— `\b` ne permet de reconnaître que les caractères qui sont aux bornes des mots (un mot étant une suite de caractères alpha-numériques séparés par des caractères non alpha-numériques).

Exercice :

— Identifier tous les mots de `"ton the t'a-t-il ote ta toux?"` qui commencent par un `t`.

Ce ne sont que quelques exemples, n'hésitez pas à vous reporter à une documentation plus complète si vous avez besoin.

4.8 Modifier les chaînes

4.8.1 Découpage de chaînes

La fonction `string` est une fonction du module `re` qui permet de séparer une chaîne suivant une ER. Par exemple :

```
>>> p = re.compile('[^a-zA-Z]')
>>> p.split("Oh! un exemple, simple, de separation.")
['Oh', '', 'un', 'exemple', '', 'simple', '', 'de', 'separation', '']
>>> p.split("Oh! un exemple, simple, de separation.", 4)
['Oh', '', 'un', 'exemple', ' simple, de separation.']
```

4.8.2 Chercher et remplacer

On peut également trouver toutes les occurences d'un motif, et les remplacer par une chaîne différente. La fonction `sub()` prend en argument ce par quoi on veut remplacer l'ER, et la chaîne dans laquelle on veut le faire. On peut spécifier de manière optionnelle le nombre maximum de fois où l'on veut effectuer le remplacement (par défaut, cette valeur vaut 0 ce qui signifie toutes les occurences possibles).

Ainsi,

```
>>> p = re.compile('vert|jaune|rouge')
>>> p.sub("de couleur", "une chaussure rouge et un chapeau vert")
'une chaussure de couleur et des chaussettes de couleur'
>>> p.sub("de couleur", "une chaussure rouge et un chapeau vert", 1)
'une chaussure de couleur et un chapeau vert'
```

Exercice :

— Créer une expression régulière telle que les trois adjectifs de couleur soient remplacés qu'ils soient au singulier ou au pluriel.

Appendice

Conversion en string

Une façon très pratique de convertir des valeurs en `str` est d'utiliser les abbréviations de format avec %. La syntaxe est d'entrer une `string` dans laquelle les valeurs à écrire sont remplacées par les **formats** dans lesquels elles seront écrites, puis d'entrer après la `string` la suite des valeurs à écrire. Par exemple :

```
>>> '%d'%(3)                          # écriture d'un entier
>>> '%d/%d vaut %f'%(3,7,float(3)/7) # des entiers puis un float
>>> '%.4f'%(1.0/7)                    # un flottant à 4 décimales
>>> '%e'%(1.0/700)                    # écriture scientifique
>>> '%8g' %(10000000.0/7)             # écriture scientifique ou non
>>>                                   #  qui tient sur 8 caractères
>>> '%4g'%(1.0)
>>> '%-4g'%(1.0)
>>> n='toto'; i=4
>>> '%s*%d=%s'%(n,i,i*n)              # string, int, string
```

Aide sur les modules

On peut lister tous les modules disponibles en entrant dans l'interface d'aide. Exemple :

```
>>> help()          # le prompt change car on rentre dans l'aide
help> modules         # liste complete
help> math            # aide du module math (taper q pour en sortir)
help> q
>>>
```